miniSAM: A Flexible Factor Graph Non-linear Least Squares Optimization Framework

Jing Dong[1] and Zhaoyang Lv[2]

***Abstract*— Many problems in computer vision and robotics can be phrased as non-linear least squares optimization problems represented by *factor graphs*, for example, simultaneous localization and mapping (SLAM), structure from motion (SfM), motion planning, and control. We have developed an open-source C++/Python framework *miniSAM*, for solving such factor graph based least squares problems. Compared to most existing frameworks for least squares solvers, miniSAM has (1) full Python/NumPy API, which enables more agile development and easy binding with existing Python projects, and (2) a wide list of sparse linear solvers, including CUDA enabled sparse linear solvers. Our benchmarking results shows miniSAM offers comparable performances on various types of problems, with more flexible and smoother development experience.**

I. Introduction

Solving non-linear least squares is important to many areas in robotics, including SLAM [1], SfM [2], motion planning [3], and control [4], [5]. Furthermore, researchers in these areas often use *factor graphs*, a probabilistic graphical representation to model the non-linear least squares problem. Dellaert and Kaess [1] first connected factor graphs to non-linear least squares, and the graph inference algorithms to sparse linear algebra algorithms.

There are existing libraries for solving non-linear least squares problems. Existing widely used frameworks by SLAM and SfM communities include Ceres [6], g2o [7], and GTSAM [8]. In particular, GTSAM uses factor graph to model the non-linear least square problems, and solves the problems using graphical algorithms rather than sparse linear algebra algorithms. However, for performance reasons all existing frameworks are implemented in C++ and therefore have the disadvantage that they require complex C++ programing, especially when users merely want to define or customize loss functions.

We introduce a flexible, general and lightweight factor graph optimization framework *miniSAM*[†]. Like GTSAM, miniSAM uses factor graphs to model non-linear least square problems. The APIs and implementation of miniSAM are heavily inspired and influenced by GTSAM, but miniSAM is a much more lightweight framework, and that extends the flexibility of GTSAM as follows:

- Full Python/NumPy API, with the ability to define custom cost functions and optimizable manifolds to enable faster and easier prototyping.
- A wide list of sparse linear solver choices, including CUDA supported GPU sparse linear solvers.
- It is lightweight and requires minimal external dependencies, thus making it great for cross-platform compatibility.

*This work was mostly finished when both authors were PhD students at College of Computing, Georgia Institute of Technology, Atlanta, USA. We would like to thank Prof. Frank Dellaert and Dr. Mustafa Mukadam giving suggestions on this work. This work received no financial support.

[1]thu.dongjing@gmail.com

[2]zhaoyang.lv@gatech.edu

[†]https://github.com/dongjing3309/minisam

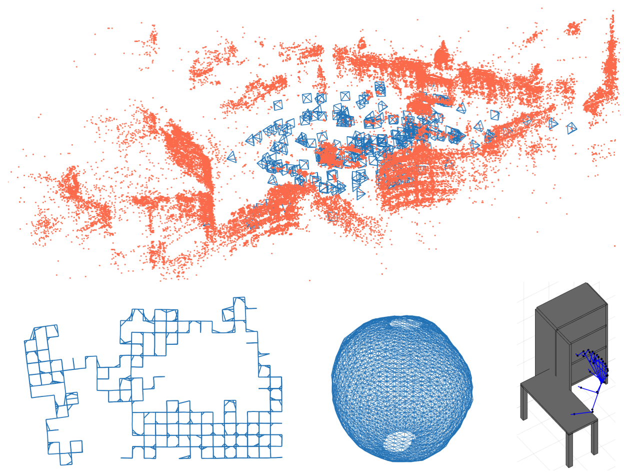


Fig. 1: Example problems solved by miniSAM. Top: bundle adjustment problem `Trafalgar` [9], camera poses are shown in blue and landmarks are shown in red. Bottom (from left to right): 2D pose graph problem `M3500` [10], 3D pose graph problem `Sphere` [10], Barrett WAM arm motion planning problem [3].

In this paper, we first give an introduction to non-linear least squares and the connection between sparse least squares and factor graphs. Then, we introduce the features and basic usage of miniSAM, finally we show benchmarking results of miniSAM on various SLAM problems.

II. Introduction to Non-linear Least Squares and Factor Graphs

A. Non-linear Least Square Optimization

Non-linear least squares optimization is defined by

$$x^* = \operatorname*{argmin}_x \sum_i \rho_i\big(\parallel f_i(x) \parallel^2_{\Sigma_i} \big), \tag{1}$$

where $x \in \mathcal{M}$ is a point on a general n-dimensional manifold, $x^* \in \mathcal{M}$ is the solution, $f_i \in \mathbb{R}^m$ is a m-dimensional vector-valued error function, ρ_i is a robust kernel function, and $\Sigma_i \in \mathbb{R}^{m\times m}$ is a covariance matrix. The Mahalanobis distance is defined by $\parallel v \parallel^2_\Sigma \doteq v^T \Sigma^{-1} v$ where $v \in \mathbb{R}^m$ and Σ^{-1} is the information matrix. If we factorize the information matrix by Cholesky factorization $\Sigma^{-1} = R^T R$, where R is upper triangular, we have

$$\| v \|_\Sigma^2 = v^T \Sigma^{-1} v = v^T R^T R v = \| Rv \|^2 . \tag{2}$$

If we consider the simplified case where ρ_i is identity and define $h_i(x) \doteq R_i f_i(x)$, then Eq. (1) is equivalent to

$$x^* = \underset{x}{\operatorname{argmin}} \sum_i \| h_i(x) \|^2 \tag{3}$$

as per Eq. (2). If we define a *linearization point* $x_0 \in \mathcal{M}$, and the Jacobian matrix of $h_i(x)$

$$J_i \doteq \frac{\partial h_i(x)}{\partial x}\Big|_{x=x_0} \tag{4}$$

then the Taylor expansion is given by

$$h_i(x_0 + \Delta x) = h_i(x_0) + J_i \Delta x + O(\Delta x^2), \tag{5}$$

which we can use to solve the least square problem by searching a *local* region near x_0, and find the solution by iteratively solving a *linearized* least squares problem

$$\Delta x^* = \underset{\Delta x}{\operatorname{argmin}} \sum_i \| J_i \Delta x + h_i(x_0) \|^2, \tag{6}$$

where $\Delta x^* \in \mathbb{R}^n$, and the solution is updated by

$$x^* = x_0 + \Delta x^*. \tag{7}$$

If $\mathcal{M}$ is simply a vector space $\mathbb{R}^n$ then the above procedure is performed iteratively in general by setting x_0 of next iteration from x^* of current iteration, until x^* converges. Trust-region policies like Levenberg-Marquardt can be also applied when looking for Δx^*.

When $\mathcal{M}$ is a general manifold, we need to define a local coordinate chart of $\mathcal{M}$ near x_0, which is an invertible map between a local region of $\mathcal{M}$ around x_0 and the local Euclidean space, and also an operator $\oplus$ that maps a point in local Euclidean space back to $\mathcal{M}$. Thus Eq. (7) on general manifolds is

$$x^* = x_0 \oplus \Delta x^*. \tag{8}$$

A simple example of $\oplus$ is for the Euclidean space where it is simply the plus operator.

To solve the linear least squares problem in Eq. (6), we first rewrite Eq. (6) as

$$\Delta x^* = \underset{\Delta x}{\operatorname{argmax}} \| J \Delta x + b \|^2, \tag{9}$$

where J is defined by stacking all J_i vertically, similarly b is defined by stacking all $h_i(x_0)$ vertically. Cholesky factorization is commonly used solve Eq. (9). Since the solution of linear least squares problem in Eq. (9) is given by the normal equation

$$J^T J \Delta x^* = J^T b, \tag{10}$$

we apply Cholesky factorization to symmetric $J^T J$, and we have $J^T J = R^T R$ where R is upper triangular. Then solving Eq. (10) is equivalent to solving both

$$R^T y = J^T b \tag{11}$$

$$R \Delta x^* = y \tag{12}$$

in two steps, which can be both solved by back-substitution given that R is triangular. Other than Cholesky factorization, QR and SVD factorizations can be also used to solve Eq. (9), although with significantly slower speeds. Iterative methods like pre-conditioned conjugate gradient (PCG) are also widely used to solve Eq. (10), especially when $J^T J$ is very large.

B. Connection between Factor Graphs and Sparse Least Squares

Dellaert and Kaess [1] have shown factor graphs have a tight connections with non-linear least square problems. A factor graph is a probabilistic graphical model, which represents a joint probability distribution of all factors

$$p(x) \propto \prod_i p_i(x_i), \tag{13}$$

where $x_i \subseteq x$ is a subset of variables involved in factor i, $p(x)$ is the overall distribution of the factor graph, and $p_i(x_i)$ is the distribution of each factor. The maximum a posteriori (MAP) estimate of the graph is

$$x^* = \underset{x}{\operatorname{argmax}}\, p(x) = \underset{x}{\operatorname{argmax}} \prod_i p_i(x_i). \tag{14}$$

If we consider the case where each factor has Gaussian distribution on $f_i(x_i)$ with covariance Σ_i,

$$p_i(x_i) \propto \exp\big(-\frac{1}{2} \| f_i(x_i) \|_{\Sigma_i}^2 \big), \tag{15}$$

then MAP inference is

$$x^* = \underset{x}{\operatorname{argmax}} \prod_i p_i(x_i) = \underset{x}{\operatorname{argmax}} \log\big(\prod_i p_i(x_i) \big), \tag{16}$$

$$= \underset{x}{\operatorname{argmin}} \prod_i -\log\big(p_i(x_i)\big) = \underset{x}{\operatorname{argmin}} \sum_i \| f_i(x_i) \|_{\Sigma_i}^2 . \tag{17}$$

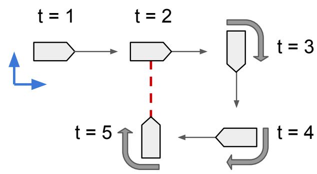
The MAP inference problem in Eq. (17) is converted to the same non-linear least squares optimization problem in Eq. 1, which can be solved following the same steps in Section II-A.

There are several advantages of using factor graph to model the non-linear least squares problem in SLAM. Factor graphs encode the probabilistic nature of the problem, and easily visualize the underlying sparsity of most SLAM problems since for most (if not all) factors x_i are very small sets. We give an example in the next section, which clearly visualizes this sparsity in a factor graph.

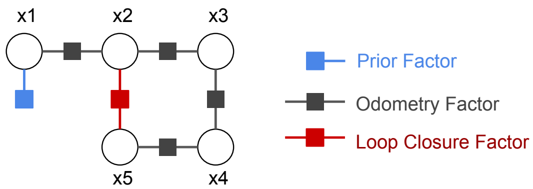
C. Example: A Pose Graph

Here we give a simple example of using factor graph to solve a small pose graph problem. The problem is shown in Fig. 2a, where a vehicle moves forward on a 2D plane, makes a 270 degrees right turn, and has a relative pose loop closure measurement which is shown in red. If we want to estimate the vehicle's poses at times $t = 1, 2, 3, 4, 5$, we define the system's state variables

$$x = \{x_1, x_2, x_3, x_4, x_5\}, \tag{18}$$



(a) Problem



(b) Factor graph

Fig. 2: An example 2D pose graph problem, with pose variables (shown as white circles), a prior (shown as a blue factor), four odometry measurements (shown as black factors) and a single loop closure measurement (shown as a red factor).

where $x_i \in SE(2), i = 1, 2, 3, 4, 5$ is the vehicle pose at $t = i$. Then the factor graph models the pose graph problem as

$$p(x) \propto \underbrace{p(x_1)}_{\text{prior}} \underbrace{p(x_1, x_2)p(x_2, x_3)p(x_3, x_4)p(x_4, x_5)}_{\text{odometry}} \underbrace{p(x_2, x_5)}_{\text{loop closure}} \tag{19}$$

which is shown in Fig. 2b. As shown in Eq. (19) and Fig. 2b, there are three types of factors: (1) A prior factor, which gives a prior distribution on first pose, and locks the solution to a world coordinate frame. (2) Odometry factors, which encode the relative poses odometry measurements between $t = i$ and $t = i + 1$. (3) A loop closure factor, which encodes the relative poses measurement between $t = 2$ and $t = 5$.

The sparsity of the example problem is clearly shown by the factor graph: all the factors are unary or binary. This is actually true for all pose graph optimization problems. We can further show the sparsity of the underlying linear system we solve in Eq. (9) and Eq. (10)

$$J = \begin{pmatrix} J_{11} & & & & \\ \hline J_{21} & J_{22} & & & \\ & J_{32} & J_{33} & & \\ & & J_{43} & J_{44} & \\ & & & J_{54} & J_{55} \\ \hline & J_{62} & & & J_{65} \end{pmatrix} \begin{matrix} \text{prior} \\ \\ \text{odometry} \\ \\ \\ \text{loop closure} \end{matrix}, \tag{20}$$

$$H = J^T J = \begin{pmatrix} H_{11} & H_{12} & & & \\ H_{21} & H_{22} & H_{23} & & H_{25} \\ & H_{32} & H_{33} & H_{34} & \\ & & H_{43} & H_{44} & H_{45} \\ & H_{52} & & H_{54} & H_{55} \end{pmatrix}. \tag{21}$$

We can clearly see both J and $J^T J$ have block-wise sparse structures.

TABLE I: Features comparison across frameworks.

	Ceres	g2o	GTSAM	miniSAM
C++: Custom cost/factor	•	•	•	•
C++: Numerical jacobian	•	•		•
C++: Auto-diff jacobian	•	•	•	
C++: Custom manifold	•	•	•	•
MATLAB: API binding			•	
Python: API binding	•[1]	•[1]	•	•
Python: Custom cost/factor				•
Python: Numerical jacobian				•
Python: Auto-diff jacobian				
Python: Custom manifold				•

[1]by third-party.

III. Solving Non-linear Least Square Problems with miniSAM

In this section we give some details about our miniSAM library and the basic introduction of how to use miniSAM. miniSAM is implemented with C++11, and provides both native C++ API and Python API provided by pybind11 [11]. miniSAM is a very lightweight library: the core implementation has only 8k lines of C++ code, plus 4k lines of test code and 2k lines of Python wrapper code. Also miniSAM requires minimal external dependencies (only Eigen [12] linear algebra library is required). This makes miniSAM great for cross-platform compatibility. Currently miniSAM compiles with most major compliers (GCC, Clang, Microsoft Visual C++) on most major OSs (Linux, macOS, Windows).

One of the most important features of miniSAM is its high flexibility with the Python interface, which provides the ability to create custom factors and manifolds directly in Python. Although most existing frameworks provide (or by third-parties) binding to script languages (like Python and MATLAB) that enable them to define graph structures, they all lack ability to directly define factors and manifolds in script languages. To use custom factors or manifolds in script languages, users need to first define them in C++, then bind the interfaces to script languages, which is inconvenient. Table I gives a comparison of multiple features in C++/Python/MATLAB between miniSAM and existing frameworks.

In the rest of this section we discuss three major use cases of miniSAM: how to define and solve factor graphs/least square optimization problems, create custom factors/cost functions, and create custom optimizable manifolds.

A. The Pose Graph Example

Here we give an example on how to use miniSAM to solve the pose graph example discussed in Section II-C. Example Python code solving this pose graph example in Fig. 2 is in Snippet 1 in the Appendix.

In the first step we construct the factor graph. In miniSAM data structure `FactorGraph` is used as the container for factor graphs. In miniSAM each variable is indexed by a *key*, which is defined by a character and an unsigned integer (e.g. x_1). Each factor has its key list that indicates the connected variables, and its loss function that has covariance Σ_i and optional robust kernel ρ_i (Cauchy and Huber robust loss functions have been implemented in miniSAM). In the pose graph example two types of factors are used: unary `PriorFactor` and binary `BetweenFactor`.

In the second step we provide the initial variable values as the linearization point. In miniSAM variable values are stored in structure `Variables`, where each variable is indexed by its key. Finally, we call a non-linear least square solver (like Levenberg-Marquardt) to solve the problem. Result variables are returned in a `Variables` structure with status code.

B. Define Factors

Here we discuss how to define a new factor in miniSAM. As mentioned defining a new factor can be done in both C++ and Python in miniSAM, by inheriting from `Factor` base class. The implementation of a factor class includes an error function `error()` that defines $f_i(x_i)$, which returns a `Eigen::VectorXd` in C++, or a NumPy array in Python. And Jacobian matrices function `jacobians()` that defines $\partial f_i(x_i)/\partial x_i$ for each variable in x_i, which return a `std::vector<Eigen::MatrixXd>` in C++, or a list of NumPy matrices in Python. We show an example prior factor on $SE(2)$ in Python in Snippet 2.

Analytic Jacobians $\partial f_i(x_i)/\partial x_i$ is usually quite complex for non-trivial factors, and is the main bottleneck for faster prototyping. miniSAM provides a solution by inheriting from `NumericalFactor` base class, numerical $\partial f_i(x_i)/\partial x_i$ through finite differencing will be evaluated during optimization, thus saving developer's time deriving analytic Jacobians. We leave automatic differentiation for Jacobian evaluation as future work.

C. Define Optimizable Manifolds

miniSAM already has build-in support for optimizing various commonly used manifold types in C++ and Python, including Eigen vector types in C++, NumPy array in Python, and Lie groups $SO(2)$, $SE(2)$, $SO(3)$, $SE(3)$ and $Sim(3)$ (implementations provided by Sophus library [13]), which are commonly used in SLAM and robotics problems.

We can also customize manifold properties of any C++ or Python class for miniSAM. In Python this is done by defining manifold-related member functions, including `dim()` function returns manifold dimensionality, and `local()` and `retract()` functions defines the local coordinate chart. An example of defining a vector space manifold $\mathbb{R}^2$ in Python is in Snippet 3. In C++ we use a non-intrusive technique called *traits*, which is a specialization of template `minisam::traits<>` for the type we are adding manifold properties. Using traits to define manifold properties has two advantages: (1) optimizing a class without modifying it, or even without knowing details of implementation (e.g. adding miniSAM optimization support for third-party C/C++ types), (2) making optimizing primitive type (like float/double) possible.

IV. Experiments

To test the performance of miniSAM, we run a benchmark on multiple problems of different types and scales, and compare with multiple existing frameworks. We choose three SLAM and SfM problem for benchmarking, from small to large.

- 2D pose graph problem `M3500` [10], which contains 3500 2D poses and 5453 energy edges.
- 3D pose graph problem `Torus` [10], which contains 5000 3D poses and 9048 energy edges.
- Bundle adjustment problem `Dubrovnik` [9], which contains 356 camera poses, 226730 landmarks and 1255268 image measurements.

For all problems we use Levenberg-Marquardt algorithm to solve, and fix the number of iterations to 5.

We run the benchmark with the following frameworks and sparse linear solvers

- Ceres [6] with Eigen simplicial LDLT solver, and CHOLMOD [14] Cholesky solver.
- g2o [7] with Eigen simplicial LDLT solver, and CHOLMOD Cholesky solver.
- GTSAM [8] with built-in multi-frontal and sequential graph elimination solvers.
- miniSAM with Eigen simplicial LDLT solver, CHOLMOD Cholesky solver, and CUDA cuSOLVER GPGPU Cholesky solver.

For miniSAM, all factors and manifolds are implemented natively in C++. All frameworks in benchmarking are compiled in single-thread, except CHOLMOD and CUDA cuSOLVER solvers are compiled in multi-thread (using all 12 available CPU threads during benchmarking, and GPU is used with CUDA). The benchmarking is performed on a computer with Intel Core i7-6850K CPU, 128 GB memory, and a NVIDIA TITAN X GPU with 12GB graphic memory. The results are shown in Table. II, and are grouped by single-thread or multi-thread.

TABLE II: Optimization times in second of different frameworks with different sparse linear solvers, grouped by single-thread or multi-thread.

	2D-PG	3D-PG	BA
Ceres + Eigen LDLT	0.090	2.735	54.96
g2o + Eigen LDLT	0.059	2.697	63.66
GTSAM + Multifrontal Cholesky	0.228	2.002	83.67
GTSAM + Sequential Cholesky	0.207	2.836	83.85
miniSAM + Eigen LDLT	0.088	3.341	64.38
Ceres + CHOLMOD	0.080	0.941	28.17
g2o + CHOLMOD	0.064	0.821	35.68
miniSAM + CHOLMOD	0.090	1.107	39.24
miniSAM + cuSOLVER Cholesky	0.458	1.791	49.77

We can see in Table. II that when the same sparse linear solver is used, miniSAM has slightly worse runtime compare to Ceres and g2o, but (except for 3D pose graph case) has better runtime compared to GTSAM, which does not use third-party sparse linear solvers. The extra overhead of miniSAM compare to Ceres and g2o are mainly due to two major miniSAM design choices:

- miniSAM avoids using any compile-time array or matrix, and all internal vectors and matrices are dynam-

ically allocated. The use of dynamic size arrays involves extra memory allocation overhead and forbids any compile-time optimization by modern CPU SIMD instructions.
- miniSAM avoids using any raw pointer and manual memory management.

The reason to make above design choices is that to make miniSAM have a Python API consistent with C++ API, and to make Python interface possible to implement, since Python does not have machinery to support template programming or explicit memory management.

We also found CUDA cuSOLVER is not as fast as CHOLMOD CPU solver when using all 12 available CPU threads, and it is particularly slow on small problems. Finally, CUDA cuSOLVER has an one-time launch delay of about 350ms, once per executable launch. Given such circumstances using CUDA cuSOLVER is currently only good for large problems.

V. Conclusion

We gave a brief introduction to miniSAM, our non-linear least squares optimization library. We demonstrate the basic usage of miniSAM, show its flexibility in fast prototyping in Python, and its performance in benchmarking of multiple types of problems in SLAM and robotics applications. We recognize miniSAM has a relatively small performance loss compared to other state-of-the-art frameworks, mostly due to miniSAM's design to adapt Python API, so currently miniSAM is not great for performance-critical applications. But hopefully we can solve the problem in the future by porting better sparse linear solvers (like GPU-enabled iterative solver) to mitigate this issue.

References

[1] F. Dellaert and M. Kaess, "Square Root SAM: Simultaneous localization and mapping via square root information smoothing," *Intl. J. of Robotics Research*, vol. 25, pp. 1181–1203, Dec 2006.

[2] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon, "Bundle adjustment – a modern synthesis," in *Vision Algorithms: Theory and Practice* (W. Triggs, A. Zisserman, and R. Szeliski, eds.), vol. 1883 of *LNCS*, pp. 298–372, Springer Verlag, 2000.

[3] J. Dong, M. Mukadam, F. Dellaert, and B. Boots, "Motion planning as probabilistic inference using Gaussian processes and factor graphs," in *Robotics: Science and Systems (RSS)*, 2016.

[4] D.-N. Ta, M. Kobilarov, and F. Dellaert, "A factor graph approach to estimation and model predictive control on unmanned aerial vehicles," in *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 181–188, 2014.

[5] M. Mukadam, C.-A. Cheng, X. Yan, and B. Boots, "Approximately optimal continuous-time motion planning and control via probabilistic inference," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 664–671, 2017.

[6] S. Agarwal, K. Mierle, and Others, "Ceres solver." http://ceres-solver.org.

[7] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g2o: A general framework for graph optimization," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, (Shanghai, China), May 2011.

[8] F. Dellaert, "Factor graphs and GTSAM: A hands-on introduction," Tech. Rep. GT-RIM-CP&R-2012-002, Georgia Institute of Technology, 2012.

[9] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski, "Bundle adjustment in the large," in *European Conf. on Computer Vision (ECCV)*, 2010.

[10] L. Carlone, R. Tron, K. Daniilidis, and F. Dellaert, "Initialization techniques for 3D SLAM: a survey on rotation estimation and its use in pose graph optimization," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 4597–4604, 2015.

[11] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between C++11 and python," 2017. https://github.com/pybind/pybind11.

[12] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http://eigen.tuxfamily.org, 2010.

[13] H. Strasdat, "Sophus: C++ implementation of lie groups using eigen." https://github.com/strasdat/Sophus.

[14] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 22, 2008.

APPENDIX: PYTHON EXAMPLE CODE SNIPPETS

Snippet 1. A pose graph optimization in Python

```
import numpy as np
from minisam import *
from minisam.sophus import *

# build factor graph for least square problem
graph = FactorGraph()
loss = DiagonalLoss.Sigmas(np.array([1.0, 1.0, 0.1])) # loss function of sensor measurement model
graph.add(PriorFactor(key('x', 1), SE2(SO2(0), np.array([0, 0])), loss)) # prior as first pose
graph.add(BetweenFactor(key('x', 1), key('x', 2), SE2(SO2(0), np.array([5, 0])), loss)) # odometry measurements
graph.add(BetweenFactor(key('x', 2), key('x', 3), SE2(SO2(-3.14/2), np.array([5, 0])), loss))
graph.add(BetweenFactor(key('x', 3), key('x', 4), SE2(SO2(-3.14/2), np.array([5, 0])), loss))
graph.add(BetweenFactor(key('x', 4), key('x', 5), SE2(SO2(-3.14/2), np.array([5, 0])), loss))
graph.add(BetweenFactor(key('x', 5), key('x', 2), SE2(SO2(-3.14/2), np.array([5, 0])), loss)) # loop closure

# variables initial guess, with random added-on noise
init_values = Variables()
init_values.add(key('x', 1), SE2(SO2(0.2), np.array([0.2, -0.3])))
init_values.add(key('x', 2), SE2(SO2(-0.1), np.array([5.1, 0.3])))
init_values.add(key('x', 3), SE2(SO2(-3.14/2 - 0.2), np.array([9.9, -0.1])))
init_values.add(key('x', 4), SE2(SO2(-3.14 + 0.1), np.array([10.2, -5.0])))
init_values.add(key('x', 5), SE2(SO2(3.14/2 - 0.1), np.array([5.1, -5.1])))

# solve least square optimization by Levenberg-Marquardt algorithm
opt = LevenbergMarquardtOptimizer()
result_values = Variables() # results
status = opt.optimize(graph, init_values, result_values)
if status != NonlinearOptimizationStatus.SUCCESS:
    print("optimization error :", status)
```

Snippet 2. A minimal Python prior factor example on SE(2)

```
import numpy as np
from minisam import *

# python implementation of prior factor on SE2
class PyPriorFactorSE2(Factor): # or inherit from NumericalFactor
    # constructor
    def __init__(self, key, prior, loss):
        Factor.__init__(self, 3, [key], loss)
        self.prior_ = prior
    # make a deep copy
    def copy(self):
        return PyPriorFactorSE2(self.keys()[0], self.prior_, self.lossFunction())
    # error vector
    def error(self, variables):
        curr_pose = variables.at(self.keys()[0]) # current variable
        return (self.prior_.inverse() * curr_pose).log()
    # jacobians, not needed if inherit from NumericalFactor
    def jacobians(self, variables):
        return [np.eye(3)]
```

Snippet 3. A minimal Python 2D point optimizable manifold

```
import numpy as np

# A 2D point class (x, y)
class PyPoint2D(object):
    # constructor
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)
    # local coordinate dimension
    def dim(self):
        return 2
    # map manifold point other to local coordinate
    def local(self, other):
        return np.array([other.x - self.x, other.y - self.y], dtype=np.float)
    # apply changes in local coordinate to manifold, \oplus operator
    def retract(self, vec):
        return PyPoint2D(self.x + vec[0], self. + vec[1])
```